Advanced skilearn

Building Pipelines with Skilearn

SimpleAI.ch

Basic pipeline

```
>>> from sklearn import neighbors, datasets, preprocessing
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.metrics import accuracy_score
```

Load skilearn example dataset

```
>>> iris = datasets.load_iris()
>>> X, y = iris.data[:, :2], iris.target
```

Prepare the data

```
# split to train/test
>>> X_train, X_test, y_train, y_test = train_test_split(
                    X, y, train_size=0.7, random_state=0)
# scale input data
>>> scaler = preprocessing.StandardScaler().fit(X_train)
>>> X_train = scaler.transform(X_train)
>>> X_test = scaler.transform(X_test)
```

Create classifier & train it

```
>>> knn = neighbors.KNeighborsClassifier(n_neighbors=5)
>>> knn.fit(X_train, y_train)
```

Predict test values, & measure error

```
>>> y_pred = knn.predict(X_test)
>>> accuracy_score(y_test, y_pred)
```

Organize multiple steps in a pipeline

Create pipeline with defined steps

Pipeline

encapsulate multiple steps with pipeline

```
# each step requires a name, and a function
>>> pipe = Pipeline([
            ('scaler', preprocessing.StandardScaler()),
            ('knn',neighbors.KNeighborsClassifier(n_neighbors=5))
           ])

# Multiple preprocessing steps can be added
>>> pipe = Pipeline([
            ('transform_1', preprocessing.function_1()),
            ('transform_1', preprocessing.function_1()),
            ('transform_1', preprocessing.function_1()),
             ...,
            ('knn',neighbors.KNeighborsClassifier(n_neighbors=5))
           ])
```

API

```
# Use standard skileanr API with the pipeline
>>> pipe.fit(X,y)
>>> pipe.sore(X,y)
>>> pipe.predict(X,y)
>>> MAE(y, pipe.predict(X,y))
```

Control over steps & parameters

```
# Check the pipeline
>>> pipe.named_steps                    # basic text info
>>> pipe.get_params()                   # to see parameters at each step
>>> set_config(display="diagram"); pipe.  # beatiful visualizations (HTML)

# disable some steps
>>> from= Pipeline([
            ('scaler', None)
            ('knn',neighbors.KNeighborsClassifier(n_neighbors=5))
           ])

# set parameters for individual steps
''' <step_name>__<parameter_name>'''
>>> pipe(knn__n_neighbors=<value>)
```

Replace them, with one custom transformer

Add Multistep and/or Custom transformers

make_pipeline()

- Give names to steps automatically,
- Useful for pre-processing steps

FunctionTransformer()

Provides skilearn transformer API to custom functions

- ie. fit(), trasnform(), etc ...
- Allows using custom function with pipeline() & make_pieline to build more complex, mutistep tranformers/pieplines

```
# create custom transformer
'' unlike pipeline, make_pipepline
  generates names for steps automatically
  ->> lowercase name of an estimator
  otherwise, these two fucntions work in the same way
>>> import numpy as np
>>> from sklearn.pipeline import make_pipeline
>>> from sklearn.preprocessing import FunctionTransformer
>>> from sklearn.preprocessing import StandardScaler,

>>> log_scale_transformer = make_pipeline(
...                         FunctionTransformer(np.log, validate=False),
...                         StandardScaler()
...                         )
```

Add different transformers for different columns

ColumnTransformer()

Allows applying different transformers to different columns in one dataset

- List with transformer + column names (in a list)
- Option to leave some columns without changes,
  - See, "passthrough"
- Option to remove, other columns,
  - See, "drop"

```
# Create fdata preprocessor with ColumnTransformer()
''' ColumnTransformer takes a list with instrucitons for each column/col. Group.
  for each of them you must provide the folloiwing
        (i)  unique preprocessor name
        (ii). Transformer function, "passthrough" str, to not make any modiff's
              - if more then one function, you need to create it separately,
                using pieline or make pipline,
                like log_scale_transformer in this example.
        (iii) column names in input data, that will be transformed  (LIST)
'''
>>> from sklearn.compose import ColumnTransformer
>>> preprocessor = ColumnTransformer(
        transformers=[
            ("passthrough_numeric", "passthrough", ["column_1"]),
            ("binned_numeric", KBinsDiscretizer(n_bins=10), [" column_2", " col..."]),
            ("log_scaled_numeric", log_scale_transformer, [" column_4"]),
            ("onehot_categorical", OneHotEncoder(), [" column_5", " column_5"]),
        ],
        remainder="drop",     # TWO OPTION {'drop', 'passthrough'}
    )
```

If different transformers are set based on dtype in the data, you may set them automatically, With skilearn function

Tune hyperparameters with GridSearch

```
# test different k values
>>> Gs_results = []
>>> for k in list(range(1,10)):
...                    # set k in the pipeline
...                    pipe(knn__n_neighbors=i)
...                    pipe.fit(X_tr, y_tr)
..                     # collect the results
...                    gs_results.append(
                              'k':k,
                              test_mae: MAE(y_te, pipe.predict(X_te))
# convert the results to pd.dataframe & plot
>>> gs = pd.DataFrame(gs_results)
>>> plt.plot(gs['k'], gs['test_mae'])
>>> plt.xlabel,   plt.legend(),   plt.show() ....
```

Used for grid search

Report results

make_column_selector()

Can select columns inpandas df based on

(i) Pandas data frame dtypes

(ii) columns name with a regex.

CAUTION: When using multiple selection criteria, **all** criteria must match for a column to be selected.

```
>>> from sklearn.compose
...     import make_column_selector  as selector
>>> preprocessor = ColumnTransformer([
....     (num, numeric_transformer,
....     Selector(dtype_include="Category"),
```

- *Pattern*
- *dtype_include*
- *dtype_exclude*

df.info()

Used to find out how the columns will be categorized with the selector

supported Pandas df dtypes:

- ***numeric:*** *use "numeric" , np.number, 'number'*
- **categorical:** use "category"
- **object:** use object - this included str, and all other columns encoded as object (may be of any dtype)
- **datetimes:** use np.datetime64, 'datetime' or 'datetime64'
- and other .... See my notes on transformers,

ParameterGrid()

Next Slide

Advanced sklearn

TUNNING HYPERPARAMETERS

SimpleAI.ch

Basic pipeline with CV and GridSearch

Data preparation

train_test_split()

You may use more advanced options with k-fold

CV allows repeating this part on subset of data to find optimal hyperparameters

Data transformers

imputers

Model

ParameterGrid()

May also be used for some transformers and imputers

Estimator function

In many cases estimator allows selecting different error functions

Evaluation

est.predict()

sklearn.metrics()

baseline

HYPERPARAMETERS

Def: Parameters that are not directly learnt within estimators
Eg: C, kernel and gamma for Support Vector Classifier, alpha for Lasso, etc.

 # to find the names and current values for all parameters:

```
>>> estimator.get_params()
```

ParameterGrid()

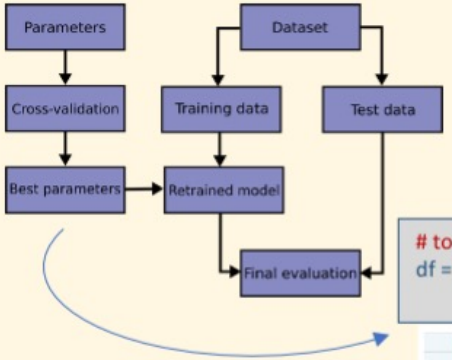
Simple GridSearch

Used to iterate over parameter value combinations

- Applies Python built-in function iter.
- The order of the generated parameter combinations deterministic.
- Can be accessed as any list or iterator
- ALL OTHER ELEMENTS OF THE PIELINE MUST BE ADDED

```
>>> from sklearn.model_selection import ParameterGrid

# option 1. takes dict with param name & their values
"""returns all combinations of these params""
>>> param_grid = {
        'a': [1, 2],
        'b': [True, False] } # list with each parameter
# may be accesses with for loop, or turn into the list
>>> list(ParameterGrid(param_grid)) == (
...   [{'a': 1, 'b': True}, {'a': 1, 'b': False},
...    {'a': 2, 'b': True}, {'a': 2, 'b': False}])

# option 2. may take several lists of dict with param. Names:values
"""it will return grid with comnbiation of all param values, in each
dictionary, but not between them !"""
>>> grid = [
            {'kernel': ['linear']},
            {'kernel': ['rbf'], 'gamma': [1, 10]
        }
# returns the following:
        list(ParameterGrid(grid)) ==
                [{'kernel': 'linear'},
...              {'kernel': 'rbf', 'gamma': 1},
...              {'kernel': 'rbf', 'gamma': 10}]
```

CROSS-VALIDATION



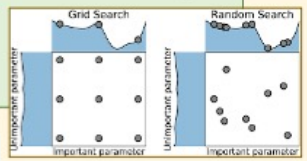

```
# to get the results in pd df
df = pd.DataFrame(.
        est.cv_results_)
```

	iter	n_resources	mean_test_score	params
0	0	125	0.983667	{'criterion': 'entropy', 'max_ 'max_features': 9, 'min_sa

BRUTE FORCE METHODS

GridSearchCV()

- **Brut Force Approach = Exhaustive Grid Search for provided parameter ranges, or values, specified with** with ParameterGrid() within that function
- Can return multiple scoring methods
- All combinations of parameters will be tested on entire provided dataset



RandomizedSearchCV()

- **Parameters:**
  - **Hyperarameter parameters sampling** - done with a dictionary, similar to specifying parameters for GridSearchCV. Subsequently, these are randomly sampled from that sampe or distribution (sampled uniformly)

```
{'C': scipy.stats.expon(scale=100),
 'gamma': scipy.stats.expon(scale=.1),
  'kernel': ['rbf'],
  'class_weight':['balanced', None]}
```

  - **N_iter** – additional, parameter controlling how many times the algorithm should samples from hypeparam. Space
- ***COMMENT:***
  - **Use Scipy.stats module -** contains many useful distributions for sampling parameters, eg: expon, gamma, uniform or randint
  - **For continuous parameters, such as C** – you must specify a continuous distribution to take full advantage of the randomization. This way, increasing n_iter will always lead to a finer search.
  - For log-uniform use:

```
>>> loguniform(1, 100)  # returns[1, 10, 100]
Or
>>> np.logspace(0, 2, num=1000)
# example:
from sklearn.utils.fixes import loguniform
{'C': loguniform(1e0, 1e3), etc...
```

SUCCESSIVE HALVING (SH)

- **Iterative approach**
- First, all candidates (the parameter combinations) are evaluated with a small amount of resources. Then, only a subset of param. Comb's are selected to next iteration.
- Each iteration is allocated an increasing amount of resources per candidate, eg. sample number.
- Typical Iteration parameters:
  - Sample number
    - Factor; >1, Default values ==2; controls the rate at which the resources grow, and the rate at which the number of candidates decreases
    - resource & min_resources
    - Eg: min_resources=10 and factor=2 params will give the following iterations in respect to sampel nr. [10, 20, 40, 80, 160, 320, 640]
  - More specific params – eg: n_estimators in a random forest
- **Best candidates are selected based on:**
  - param combinations, that have consistently ranked among the top-scoring candidates across all iterations.
  - only a subset of candidates 'survive' until the last iteration

HalvingGridSearchCV()

HalvingRandomizedSearchCV()

- ***These are Halving alternatives to brute force CV functions***
- ***Caution:*** These estimators are still **experimental**: their predictions and their API might change without any deprecation cycle. To use them, you

Functions with build-in CV

`linear_model.ElasticNetCV`(*[, l1_ratio, ...])	Elastic Net model with iterative fitting along a regularization path.
`linear_model.LarsCV`(*[, fit_intercept, ...])	Cross-validated Least Angle Regression model.
`linear_model.LassoCV`(*[, eps, n_alphas, ...])	Lasso linear model with iterative fitting along a regularization path.
`linear_model.LassoLarsCV`(*[, fit_intercept, ...])	Cross-validated Lasso, using the LARS algorithm.
`linear_model.LogisticRegressionCV`(* [, Cs, ...])	Logistic Regression CV (aka logit, MaxEnt) classifier.
`linear_model.MultiTaskElasticNetCV`(*[, ...])	Multi-task L1/L2 ElasticNet with built-in cross-validation.
`linear_model.MultiTaskLassoCV`(*[, eps, ...])	Multi-task Lasso model trained with L1/L2 mixed-norm as regularizer.
`linear_model.OrthogonalMatchingPursuitCV`(*)	Cross-validated Orthogonal Matching Pursuit model (OMP).
`linear_model.RidgeCV`([alphas, ...])	Ridge regression with built-in cross-validation.
`linear_model.RidgeClassifierCV`([alphas, ...])	Ridge classifier with built-in cross-validation.

TUNING HYPERPARAMETERS

GridSearch

Using for loops

```
# Example: test different k values
>>> Gs_results = []
>>> for k in list(range(1,10)):
...                     # set k in the pipeline
...                     pipe(knn__n_neighbors=i)
...                     pipe.fit(X_tr, y_tr)
..                      # collect the results
...                     gs_results.append(
...                                 'k':k,
...                                 test_mae: MAE(
                                            y_te, pipe.predict(X_te))
# convert the results to pd.dataframe & plot them
>>> gs = pd.DataFrame(gs_results)
>>> plt.plot(gs['k'], gs['test_mae'])
>>> plt.xlabel,   plt.legend(),   plt.show() ….
```

ParameterGrid()

Used to iterate over parameter value combinations

- Applies Python built-in function iter.
- The order of the generated parameter combinations is deterministic.
- Can be accessed as any list or iterator

```
>>> from sklearn.model_selection import ParameterGrid

# option 1. takes dict with param name & their values
"""returns all combinations of these params""
>>> param_grid = {
            'a': [1, 2],
            'b': [True, False] } # list with each parameter
# may be accesses with for loop, or turn into the list
>>> list(ParameterGrid(param_grid)) == (
...    [{'a': 1, 'b': True}, {'a': 1, 'b': False},
...     {'a': 2, 'b': True}, {'a': 2, 'b': False}])

# option 2. may take several lists of dict with param.
Names:values
"""it will return grid with comnbiation of all param values,
in each dictionary, but not between them !"""
>>> grid = [
            {'kernel': ['linear']},
            {'kernel': ['rbf'], 'gamma': [1, 10]
          }
# returns the following:
          list(ParameterGrid(grid)) ==
                    [{'kernel': 'linear'},
...                  {'kernel': 'rbf', 'gamma': 1},
...                  {'kernel': 'rbf', 'gamma': 10}]
```

AUTOml

Auto-sklearn

https://automl.github.io/auto-sklearn/master/

FEATURES

- *Allows fast classification using many popular methods*
- *It can be extended with new classification, regression and feature pre-processing methods*

CONS

- *Limited knowledge on explored space*

EXAMPLE:

```
"""has fit, predict methods"""
>>> import autosklearn.classification
>>> cls = autosklearn.classification.AutoSklearnClassifier()
>>> cls.fit(X_train, y_train)
>>> predictions = cls.predict(X_test)
```

Visualize tested parameters

Pipeline Profiler Tool

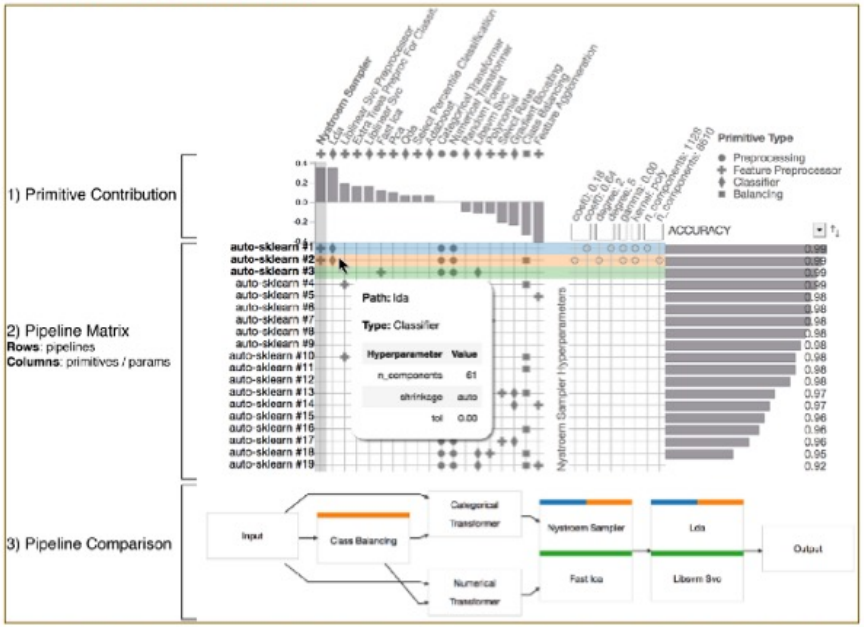
EXAMPLE

```
# classify digit dataset with automl skileanr package
>>> import sklearn.datasets
>>> import autosklearn.classification

>>> X, y = sklearn.datasets.load_digits(return_X_y=True)
>>> automl = autosklearn.classification.AutoSklearnClassifier()
>>> automl.fit(X, y, dataset_name='digits')

# Visualize results with pipeline profiler
""" it helps understanding what have happened"""
>>> import PipelineProfiler
>>> profiler_data = PipelineProfiler.import_autosklearn(automl)
>>> PipelineProfiler.plot_pipeline_matrix(profiler_data)
    # creates  the plot in the above
```

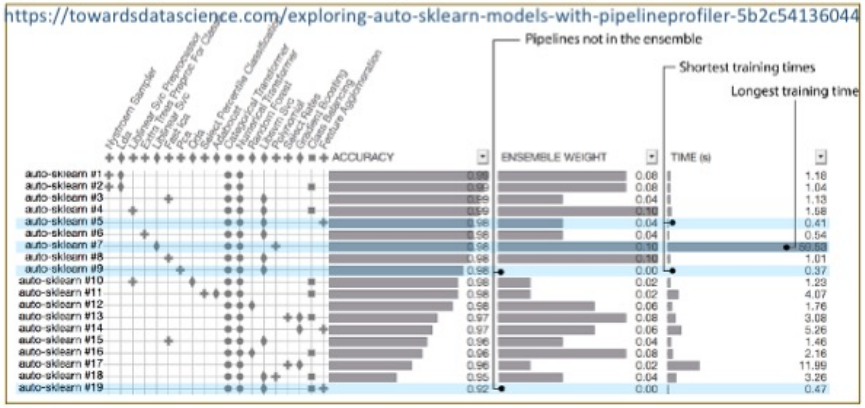
This is example of a new tool that can help with automl solutions

Using pipeline proficler we can we can see which pipeline elements:

- *were corelated with high accuracy scores*
- *contributed to long training time*
- *And many more*



https://towardsdatascience.com/exploring-auto-sklearn-models-with-pipelineprofiler-5b2c54136044



Data Preprocessing

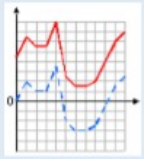
SKLEARN TRANFORMERS – PART 1

SimpleAI.ch

- FOR SCALING & CENTERING DATA FEATURES -

LINEAR TRANFORMATIONS

Each feature/column is transomed separately



Linear tranfs. **preserve** relative distance between features, and the shape of the distribution

IN PRACTICE: we often ignore the shape of the distribution and just transform the data to center it by removing the mean value of each feature, then scale it by dividing non-constant features by their standard deviation.

Original data
2 features with different means, and probably different SD

Robust Scaling
Both features have:
- MEAN = 0
- SD = IQR

Standard Sc.
Both features have:
- MEAN = 0
- SD = 1

MinMax Scaler
Features have DIFFERENT mean and SD, but
- Min value = 0
- Max value = 1

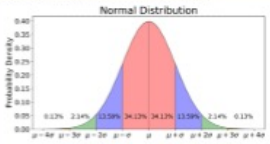
Centred around the same value , and expressed with the same unit & SD

Preserve data sparsity, and have the same range

SCALING

(i) **Used to Scale features with Gaussian distr.**
(ii) **Applied before using Gradient Descent,**
(iii) Applied before techniques that assume that assume normal distrib. Such as, linear regr., logistic regr. and linear discriminate analysis

z = (x-mean)/sd
sd = sqrt(Σ (x-mean)**2 / N)
- *Range [-∞, +∞]*
- *mean = 0*
- *sd = 1*




```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler().fit(X_train)
>>> standardized_X = scaler.transform(X_train)
>>> standardized_X_test = scaler.transform(X_test)

# you may use with_mean=False for sparse data
```

ROBUST SCALER

(i) Gives similar results as the standard scaler, but it based on median & IQR region size, instead of mean and SD.
(ii) Applied for data with **OUTLIERS**.
(iii) Can be used with different IQR size eg 5%, 25%, etc.. That may affect the results, and help scaling data to outliers.

X'i = (x-median)/IQR
- *IQR – difference between 25th and 75 percentiles (can be set as different value)*
- *Range [-∞, +∞]*
- *mean = 0*

```
>>> from sklearn.preprocessing import RobustScaler
>>> scaler = RobustScaler ().fit(X_train)
>>> standardized_X = scaler.transform(X_train)

# set different quintile range
>>> scaler = RobustScaler (quantile_range=(15, 85))
>>> scaler = RobustScaler (quantile_range=(value, 100-value)) # eg. value= 30
```

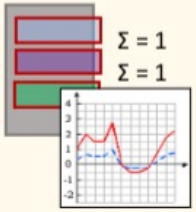
- TO SCALE SPARSE DATA -

NORMALIZATION

Scaling individual samples/rows to unit norm.
-> Sum of values in each observation (row) will be equal to 1 (unit norm).

(ii) Used for **SPARSE DATASETS** with attributes of varying scales
(iii) often used in **text classification and clustering contexts.**
(iv) alg. that use weights, eg NN, and alg, that use distance measures eg. kNN

x/(L1, L2 or Max norm)
- ***Range [0,1]***
- *Sum for each row (norm) = 1*
- ***Smoothing effect***



Normalizer

```
# {'l1', 'l2', 'max'}, default='l2
>>> from sklearn.preprocessing import Normalizer
>>> scaler = Normalizer ().fit(X_train)
>>> standardized_X = scaler.transform(X_train)
```

SCALING FEATURES TO A RANGE

Scaling features to lie between a given minimum and maximum value
eg: to convert a temperature from Celsius to Fahrenheit.

- **SPARSE DATA, especially MinMaxScaller** was designed for it.
  - Provide robustness to very small standard deviations of features
  - Do not centre values, and preserve zero entries in sparse data.
- **OUTLIERS**, but in that case there can be a problem (see caution)

Min-Max Scaler

Rescales features between 0-1

(i) Applied before using optimization alg. like **gradient descent**, that assume all values have 0-1 range
(ii) Used with alg. that use weight inputs; regression, NN.
(iii) and with alg. that use distance measures, k-NN

x' = (x – min) / (max – min)
- ***Range [0,1]***
- *If x = min(X), then x'=0*
- *If x = max(X), then x'=1*

CAUTION: may rescale values to very small intervals, if **outliers** are present

```
>>> from sklearn.preprocessing import MinMaxScaler
>>> transformer = MinMaxScaler()
>>> rescaledX = transformer.fit_transform(X)

# you may provide explicirt min/max values
>>> transformer = MinMaxScaler(feature_range=(0, 123))
```

MaxAbs Scaler

Rescales features between -1 & 1
Divides each value by maximum absolute value of each feature.

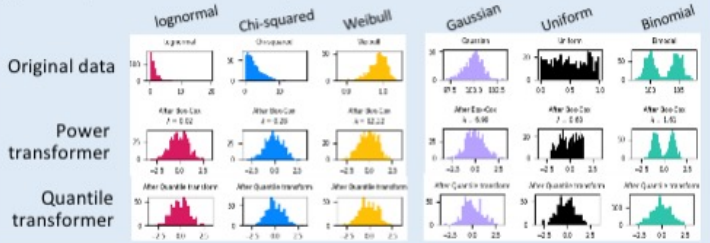
Range [-1,..,1]
NaN - remain NaN

```
>>> from sklearn.preprocessing import MaxAbsScaler
    X = [[ 1., -1., 2.], ... [ 2., 0., 0.], ... [ 0., 1., -1.]]
>>> transformer = MaxAbsScaler().fit(X)
>>> transformer.transform(X)
    array([[ 0.5, -1. , 1. ], [ 1. , 0. , 0. ], [ 0. , 1. , -0.5]])
```

- USED TO CORRECT SKEWNESS IN THE DISTRIBUSIONS -

NON-LINEAR TRANSFORMATIONS

(i) ***transforms the features to follow a uniform or a normal distribution***
(ii) To remove/reduced the impact of **OUTLIERS**
(iii) spread out the most frequent values



CAUTION:
(1). these methods don't scale the data to a predetermined range
(2). These methods may distort lin. corr. between var's measured at the same scale, but renders var's measured at different scales more directly comparable

Quantile transformer

Non-parametric approach

How it works?
- first, cdf is used to map the original values to a uniform distribution.
- then, these values are mapped to output distribution with quantile function.
- values below or above the fitted range will be mapped to the bounds of the output distr.

- Range: [0,1]
- spread out the most frequent values
- *transformed data is the approximation of the quantile position of the actual data*

```
>>> from sklearn.preprocessing import QuantileTransformer
>>> qt = QuantileTransformer(n_quantiles=10, random_state=0)
>>> qt.fit_transform(X)
```

	mpg	mpg_trans
0	18.0	0.286873
1	15.0	0.153652
2	18.0	0.286873
3	16.0	0.202771
4	17.0	0.239295

- **n_quantiles;** Typically large, default 1000.
- **output_distribution** *{'uniform', 'normal'},* distrib. used for the transformed data.
- **ignore_implicit_zeros** If True, zeros are not used, and stay as zeros.
- **Subsample** *int,* max nr of used samples

Power transformer

Parametric approach

- map data from any distribution to a Gaussian distribution
- stabilize variance
- minimize skewness.

Two methods:
'box-cox' - needs the data to be positive
'Yeo-Johnson' - data to be both negative and positive.

```
>>> pt = PowerTransformer(method='box-cox', standardize=False)
>>> X_lognormal = np.random.RandomState(616).lognormal(size=(3, 3))
>>> pt.fit_transform(X_lognormal)
```

Skilearn API

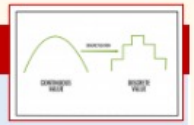
`fit`(X[, y])	Compute the median and quantiles to be used for scaling.
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`inverse_transform`(X)	Scale back the data to the original representation
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X)	Center and scale the data.

Data Preprocessing

SKLEARN TRANFORMERS – PART 2

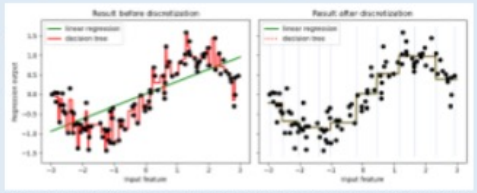
SimpleAI.ch

DISCRETISATION/BINNING



(i) Used to discretize continuous features
(ii) May improve linear models,
(iii) Reduce variance in non-linear models, like regression trees,

After discretization, linear regression and decision tree make exactly the same prediction



https://scikit-learn.org/stable/auto_examples/preprocessing/plot_discretization.html#sphx-glr-auto-examples-preprocessing-plot-discretization-py

binarization

All values > threshold are 1 and all ≤threshold are marked as 0

(i) **Applied for PROBABILITIES**
(ii) Used for Feature eng.

Generates 0 or 1 values only

```
>>> from sklearn.preprocessing import Binarizer
>>> binarizer = Binarizer(threshold=0.0).fit(X)
>>> binary_X = binarizer.transform(X_train)
```

K-bins discretization

Transforms continuous feature into a categorical feature by partitioning it into several bins within the expected value range (intervals).

- ***0 -K-1 bins***
- ***Ecoding***
  - ***One-hot-enc***
  - ***ordinal (0, 1, 2, …, k-1)***

Value	Bins
10	1
15	1
20	1
25	2
30	2

 # Encoding

- ***'onehot',: default,*** ignored features, eg missing data, novelties, are always stacked to the right
- ***'ordinal'***: return sthe bin identifier encoded as an integer value

CAUTION
Ordinal values may still be used, in most models

 # Strategy used to define the widths of the bins.

- **Uniform;** All bins in each feature have identical widths (max-min values).
- **Quantile;** All bins in each feature have the same number of points.
- **kmeansValues;** in each bin have the same nearest center of a 1D k-means cluster.

```
>>> from sklearn.preprocessing import KBinsDiscretizer
>>> enc = KBinsDiscretizer(n_bins=10, encode='onehot')
>>> X_binned = enc.fit_transform(X)
```

Sourses

https://towardsdatascience.com/5-data-transformers-to-know-from-scikit-learn-612bc48b8c89
https://scikit-learn.org/stable/modules/preprocessing.html#spline-transformer
https://scikit-learn.org/stable/auto_examples/preprocessing/plot_all_scaling.html#sphx-glr-auto-examples-preprocessing-plot-all-scaling-py

ENCODING CATEGORICAL FEATURES & TARGETS

one-hot/dummy encoding

Nan & None
- Considered as separate values,

handle_unknown='ignore'
- If ignore, these are mapped to zeros ,

drop='first'
- column with the 1st category is dropped,
- If only one cat is present, it is also dropped,
- Dropped column contains zeros, representing unknown variables (novelties)

Why to drop a column?
Done to avoid co-linearity in the input matrix in some classifiers. Eg., non-regularized regression (LinearRegression), since co-linearity would cause the covariance matrix to be non-invertible

```
>>> from sklearn.preprocessing import OneHotEncoder()
>>> enc = OneHotEncoder(
            drop='first',
            handle_unknown='ignore' )
            # unknown/novelties = will be encoded with zero
>>> enc_X = enc.fit_transform(X)
>>> enc.categories_ # returns categories,

# you may specify explicit categories with 'categories' parameter
- NOT ADVISED
 >>> X =    [['male', 'uses Safari'], ['female', 'uses Firefox']]
 >>> genders  = ['female', 'male']
 >>> browsers  = ['uses Firefox', 'uses Safari']
 >>> enc = OneHotEncoder(categories=[genders, browsers])
```

OrdinalEncoder()

transforms each categorical feature to one new feature of integers (0 to n_categories - 1)

IMPORTANT
Ordinal values can not be used directly with all scikit-learn estimators, as these expect continuous input

 # handle_unknown & unknown_value
- If =='error', an error will be raised in case of novelty is detected.
- If =='use_encoded_value', alg will use value provided with unknown_value, None, will be returned in inverse_transform

 # NaN
- Stays NaN

```
>>> from sklearn.preprocessing import OrdinalEncoder()
>>> enc = OrdinalEncoder()
>>> encoded_X = enc.fit_transform(X_train)
```

get_dummies()

One-hot encoder from pandas

https://pandas.pydata.org/docs/reference/api/pandas.get_dummies.html

Time related feature eng

https://scikit-learn.org/stable/auto_examples/applications/plot_cyclical_feature_engineering.html#sphx-glr-auto-examples-applications-plot-cyclical-feature-engineering-py

ADDING POLYNOMIAL FEATURES

PolynomialFeatures()

- **interaction_only**, if True, only the highest degree is used
- **include_bias** *If True, adds bias term, w0, with 1 only in each row.*
  For intercept

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly = PolynomialFeatures(degree=3, include_bias=True)
>>> poly.fit_transform(X)
>>> poly_get_feature_names() # will retunr names of all new features
```

with FunctionTransformer() And labda function

```
>>> poly_columns = ['col_1', 'col_2']
>>> poly_transformer = Pipeline([
            ('scaler', StandardScaler()),
            ('poly', FunctionTransformer(
                lambda X: np.c_[X, X**2, X**3])  ])
```

- **It wont add bias term !**

CUSTOM TRANFORMERS

FunctionTransformer()

Used to implement a transformer from an arbitrary function with transformer API

 # Example 1. build a transformer with a log transformation

```
>>> import numpy as np
>>> from sklearn.preprocessing import FunctionTransformer
>>> transformer = FunctionTransformer(np.log1p, validate=True)
>>> transformer.fit_transform(X)
    # validate=True; check if the imput is an array

# ensure that inverse_tranform is possible
>>> enc = transformer.fit(X, check_inverse=True)
    # if not, returns a warning
    # Use fit, before transform, to apply it.
```

 # Example 2. provide stats for twitter posts, eg. post length, and sentence nr.

```
>>> def text_stats(posts):
...         return [{'length': len(text),
...                  'num_sentences': text.count('.')}
...                  for text in posts]
>>> text_stats_transformer = FunctionTransformer(text_stats)
```

 # Example 3. just add 1 to each value in a transformed columns

```
>>> def cust_func(x):
....        return x +1
```

LABEL ENCODERS

LabelEncoder()

 # Encodes labels/targets in y,

```
>>> from sklearn.preprocessing import LabelEncoder
>>>  enc = LabelEncoder()
>>>  y = enc.fit_transform(y)
```

LabelBinarizer()

- ***Works, like one-hot encoder,***
- ***Can be used with mutivariate regression***

Data Preprocessing

IMPUTATION WITH SKLEARN

SimpleAI.ch

MISSING VALUES

DELETION — *Pandas functions/pre-processing*

- **PAIRWISE DELETIONS** — Deleting only missing data in compared cases
- **LIKEWISE DELETIONS** — dropping entire rows, with missing values
- **DROPPING COLUMNS** — dropping entire columns, with missing values

ML alg's require numeric input values, and a value to be present for each row and column in a dataset. It is common to identify missing values and replace them with a numeric value.

IMPUTATION

- **ADVANCED**
  - **kNN Based** — "nearest neighbor imputation" **kNN model** is used to predict the missing values, although any other model can also be applied — *IterativeImputer()*
  - **MICE** — **MICE; multiple imputation method used to replace missing data values** in a data set under certain assumptions about the data missingness mechanism (e.g., the data are missing at random, the data are missing completely at random) — *KNNImputer()*
- **GENERAL**
  - **Constant**
  - **Non missing = 1, Missing = 0** — *MissingIndicator()*
  - **Mean, Median, Mode, Most Frequent** — *SimpleImputer()*
- **TIME SERIES**
  - **Forward Fill**
  - **BackFill**
  - **Linear Interpolation** — *Pandas functions*

SimpleImputer()

Imputation transformer for completing missing values

- **missing_values;** defining values considered as missing in a dataset
- **Strategy;** *default='mean'*
  - **"mean"/ "median"**
  - **"most_frequent";** works for both, str, and numeric data, Caution, if >1 value, the smallest is used.
  - **"constant";** you must define **fill_value.**
- **add_indicator,** If True, a MissingIndicator transform will stack onto output of the imputer's transform. This allows a predictive estimator to account for missingness despite imputation. If a feature has no missing values at fit/train time, the feature won't appear on the missing indicator even if there are missing values at transform/test time.

```
>>> fomr sklearn.impute import SimpleImputer
>>> imp = SimpleImputer(
            missing_values=0,
            strategy='mean',
            axis=0)
>>> imp.fit_transform(X_train)
```

KNNimputer()

(i) kNN imputation (K-nearest neighbor model)

- *(i) A new sample is imputed by finding the samples in the training set "closest" to it and averages these nearby points to fill in the value.*
- *(ii) More efficient then the commonly used row average method or replacing nan with Zeros*

(ii) Alternatively: **regression models** can be used for numeric variables,

(iii) Configuration of KNN imputation involves selecting:

- (iii) **distance measure** (e.g. Euclidean)
- **(iv) k hyperparameter,** ie. the number of contributing neighbors for each prediction

```
>>> from sklearn.impute import KNNImputer
>>> imputer = KNNImputer(
...             n_neighbors=5,
...             weights='uniform',
...             metric='nan_euclidean')
...                     # nan avare method, do not use nan to compute distances,
...                     # if other methods – you must provide distance matrices,
...                       or your function
>>> imputer.fit(X)
>>> Xtrans = imputer.transform(X)
```

GOOD PRACTICE

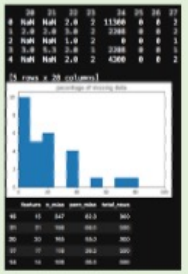
Convert missing values to np.nan in pandas.df that may have pd.NA

Example: Tuning KNNinputer

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from numpy import isnan
from sklearn.impute import KNNImputer
from sklearn.ensemble import RandomForestClassifier
from sklearn.impute import KNNImputer
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.pipeline import Pipeline
```

1. Load data, EDA on missing data points

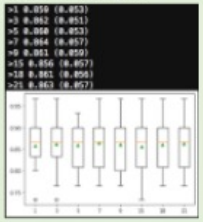
```
# load dataset
url = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/horse-colic.csv'
dataframe = pd.read_csv(url, header=None, na_values='?')

# summarize the first few rows
print(dataframe.head())

# summarize the number of rows with missing values for each column
results=[]
for i in range(dataframe.shape[1]):
    # count number of rows with missing values
    n_miss = int(dataframe[[i]].isnull().sum())
    perc = float(n_miss/dataframe.shape[0]*100)
    results.append({"feature":i, "n_miss":n_miss,
                    "perc_miss":np.round(perc,1),
                    "total_rows":dataframe.shape[0]})

# show all features on histogram
fig, ax = plt.subplots();
ax.hist(pd.DataFrame(results).perc_miss.values)
ax.set_xlim(0,100)
ax.set_title("pecentage of missing data")
plt.show();

# display columns with the highest pers/nr of missing values
pd.DataFrame(results).sort_values("perc_miss", ascending=False).head()
```



2. Find best k, suing k-fold cv, and random forest classifier

```
# split into input and output elements
data = dataframe.values
ix = [i for i in range(data.shape[1]) if i != 23]
X, y = data[:, ix], data[:, 23]

# define modeling pipeline
model = RandomForestClassifier()
imputer = KNNImputer()

# evaluate each strategy on the dataset
results = list()
strategies = [str(i) for i in [1,3,5,7,9,15,18,21]]  # k for KNNimputer
for s in strategies:
    # create the modeling pipeline
    pipeline = Pipeline(steps=[
        ('i', KNNImputer(n_neighbors=int(s))),
        ('m', RandomForestClassifier())
    ])

    # evaluate the model
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
    scores = cross_val_score(pipeline, X, y, scoring='accuracy', cv=cv, n_jobs=-1)

    # store results
    results.append(scores)
    print('>%s %.3f (%.3f)' % (s, np.mean(scores), np.std(scores)))

# plot model performance for comparison
plt.boxplot(results, labels=strategies, showmeans=True)
plt.show()
```

```
>1 0.859 (0.053)
>3 0.862 (0.051)
>5 0.860 (0.053)
>7 0.864 (0.057)
>9 0.861 (0.059)
>15 0.856 (0.057)
>18 0.861 (0.056)
>21 0.863 (0.057)
```



3. Create pipeline with selected steps& hyperparameters, fit it, and use for prediction for new data

```
# define new data
row = [2, 1, 530101, 38.50, 66, 28, 3, 3,
       np.nan, 2, 5, 4, 4, np.nan, np.nan, np.nan,
       3, 5, 45.00, 8.40, np.nan, np.nan,
       2, 11300, 00000, 00000, 2]

# create the modeling pipeline
pipeline = Pipeline(steps=[
    ('i', KNNImputer(n_neighbors=21)),
    ('m', RandomForestClassifier())
])

# fit the model
pipeline.fit(X, y)

# make a prediction
yhat = pipeline.predict([row])
print('Predicted Class: %d' % yhat[0])
```

```
Predicted Class: 2
```

The plot suggest that there is not much difference in the k value when imputing the missing values, with minor fluctuations around the mean performance (green triangle).

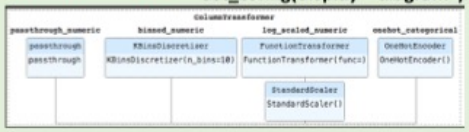
Example taken from: https://machinelearningmastery.com

APPLYING CUSTOM TRANSFORMERS

EXAMPLE: APPLYING CUSTOM TRANFORMERS

```
>>> import numpy as np
>>> from sklearn.pipeline import make_pipeline
>>> from sklearn.preprocessing import FunctionTransformer
>>> from sklearn.preprocessing import OneHotEncoder, StandardScaler, KBinsDiscretizer
>>> from sklearn.compose import ColumnTransformer

# 1. DEFINE CUTOM TRANFROMERS
# create custom transformer
'' unlike pipeline, make_pipepline
  generates names for steps automatically
  ->> lowercase name of an estimator
  otherwise, these two fucntions work in the same way
'''
>>> log_scale_transformer = make_pipeline(
...                         FunctionTransformer(np.log, validate=False),
...                         StandardScaler()
...                         )

# 2. DEFINE PREPROCESSING FUNCTION WITH DIFFERENT  TRANFORMATIONS
      FOR DIFFERENT COLUMNS
# Create final preprocessor for the data, with ColumnTransformer()
''' ColumnTransformer takes a list with instrucitons for each column/col. Group.
  for each of them you must provide the folloiwing
          (i)  unique preprocessor name
          (ii). Transformer function, "passthrough" str, to not make any modifications
                    - if more then one function, you need to create it separately,
                      using pieline or make pipline,
                      like log_scale_transformer in this example.
          (iii) column names in input data, that will be transformed  (LIST)
'''
linear_model_preprocessor = ColumnTransformer(
  transformers=[

    ("passthrough_numeric", "passthrough", ["column_1"]),

    ("binned_numeric", KBinsDiscretizer(n_bins=10), [" column_2", " column_ 3"]),

    ("log_scaled_numeric", log_scale_transformer, [" column_4"]),

    ("onehot_categorical", OneHotEncoder(), [" column_5", " column_5", " column_6"]),
  ],
  remainder="drop",     # TWO OPTION {'drop', 'passthrough'}
)
```

set_config(display="diagram")



When to use pipeline vs make_pipeline?

With Pipeline:
- Step/tranformer names are explicit,
- Ie. the name wotn't change if you change estimator/transformer used in a step,
  - e.g. if you replace LogisticRegression() with LinearSVC() you can still use clf__C.

With make_pipeline:
- shorter in use
- names are auto-generated using a straightforward rule (lowercase name of an estimator).
- Cons: if you change the function, names inside will change, an some other functions may stop working in the pipeline,

Thus:
- Use make_pipeline for quick experiments and
- Use Pipeline for more stable code/larger project
- not a big deal to use make_pipeline/Pipeline interchangeably

make_pipeline()

Here used to create multistep transformer
- Give names to steps automatically,
- Useful for pre-processing steps

FunctionTransformer()

Provides skilearn transformer API to custom functions
- ie. fit(), trasnform(), etc ...
- Allows using custom function with pipeline() & make_pieline to build more complex, mutistep tranformers/pieplines

HOW TO USE IT?

ColumnTransformer()

for: MIXED TRANFORMER TYPES

Allows applying different transformers to different columns in one dataset
- List with tranformer + column names (in a list)
- Option to leave some columns without chnages,
  - See, **"passthrough"**
- Option to remove, other columns,
  - See, **"drop"**

HOW TO BUILD CUSTOM TRANFORMERS

HOW TO USE IT?

```
# Example 1.  build a transformer with a log transformation
>>> import numpy as np
>>> from sklearn.preprocessing import FunctionTransformer
>>> transformer = FunctionTransformer(np.log1p, validate=True)
>>> transformer.fit_transform(X)
    # validate=True; check if the imput is an array

# ensure that inverse_tranform is possible
>>> enc = transformer.fit(X, check_inverse=True)
    # if not, returns a warning
    # Use fit, before transform, to apply it.

# Example 2. provide stats for twitter posts, eg. post length, and sentence nr.
>>> def text_stats(posts):
...         return [{'length': len(text),
...                  'num_sentences': text.count('.')}
...                  for text in posts]
>>> text_stats_transformer = FunctionTransformer(text_stats)

# Example 3. just add 1 to each value in a transformed columns
>>> def cust_func(x):
....        return x +1
```

SMALL PIPELINE WITH ESTIMATOR

```
>>> from sklearn import neighbors, datasets, preprocessing
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.metrics import accuracy_score

# load the data
>>> iris = datasets.load_iris()
>>> X, y = iris.data[:, :2], iris.target

# split to train/test
>>> X_train, X_test, y_train, y_test = train_test_split(
                                X, y, train_size=0.7, random_state=0)

# scale input data
>>> scaler = preprocessing.StandardScaler().fit(X_train)
>>> X_train = scaler.transform(X_train)
>>> X_test = scaler.transform(X_test)

# create classifier
>>> knn = neighbors.KNeighborsClassifier(n_neighbors=5)
>>> knn.fit(X_train, y_train)

# predict & test
>>> y_pred = knn.predict(X_test)
>>> accuracy_score(y_test, y_pred)
```

Data Preprocessing

APPLYING MIXED TYPE TRANSFORMERS

SimpleAI.ch

THE SIMPLEST PIEPLINE WITH MIXED TRANFORMER TYPES

```
# Author: Pedro Morales <part.morales@gmail.com>
#
# License: BSD 3 clause

from __future__ import print_function

import pandas as pd
import numpy as np

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split, GridSearchCV

np.random.seed(0)

# Read data from Titanic dataset.
titanic_url = ('https://raw.githubusercontent.com/amueller/'
               'scipy-2017-sklearn/091d371/notebooks/datasets/titanic3.csv')
data = pd.read_csv(titanic_url)

# We will train our classifier with the following features:
# Numeric Features:
# - age: float.
# - fare: float.
# Categorical Features:
# - embarked: categories encoded as strings {'C', 'S', 'Q'}.
# - sex: categories encoded as strings {'female', 'male'}.
# - pclass: ordinal integers {1, 2, 3}.

# We create the preprocessing pipelines for both numeric and categorical data.
numeric_features = ['age', 'fare']
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())])

categorical_features = ['embarked', 'sex', 'pclass']
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='constant', fill_value='missing')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))])

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)])

# Append classifier to preprocessing pipeline.
# Now we have a full prediction pipeline.
clf = Pipeline(steps=[('preprocessor', preprocessor),
                      ('classifier', LogisticRegression(solver='lbfgs'))])

X = data.drop('survived', axis=1)
y = data['survived']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

clf.fit(X_train, y_train)
print("model score: %.3f" % clf.score(X_test, y_test))
```

1. **Define names of eg. categorical & numeric features**
   - *List[] with feature names*
2. **Define tranformation steps for each feature type**
   - *Pipepline(steps=[])*
   - *make_pipeline()*
   - *FunctionTransformer()*
3. **Create Preprocessor With mini-pipelines for different feature types**
   - *ColumnTranformer()*
4. **Add preprocessor to larger pipeline, that also contains the model to fit**
   - *Pipepline(steps=[("prep", preprocessor()), ("clasif", ml_alg_name())])*
5. **Prepare data (train/test split) Fit the model & Evaluate it**
   - *X,y = input_data*
   - *train_test_split(x,y , test_size=0.3)*
   - *Pipeline.fit(x_train, y_train)*
   - *Pipeline.score(X_test, y_Test)*
   - *Piepline.predict(X_test, y_Test)*

Similar example, provided by skilearn with some options for automated interactions with input data in pandas dataframe

Column Transformer with Mixed Types

This example illustrates how to apply different preprocessing and feature extraction pipelines to different subsets of features, using `ColumnTransformer`. This is particularly handy for the case of datasets that contain heterogeneous data types, since we may want to scale the numeric features and one-hot encode the categorical ones.

In this example, the numeric data is standard-scaled after mean-imputation, while the categorical data is one-hot encoded after imputing missing values with a new category (`'missing'`).

In addition, we show two different ways to dispatch the columns to the particular pre-processor: by column names and by column data types.

Finally, the preprocessing pipeline is integrated in a full prediction pipeline using `Pipeline`, together with a simple classification model.

```
# Author: Pedro Morales <part.morales@gmail.com>
#
# License: BSD 3 clause

import numpy as np

from sklearn.compose import ColumnTransformer
from sklearn.datasets import fetch_openml
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split, GridSearchCV

np.random.seed(0)

# Load data from https://www.openml.org/d/40945
X, y = fetch_openml("titanic", version=1, as_frame=True, return_X_y=True)

# Alternatively X and y can be obtained directly from the frame attributes:
# X = titanic.frame.drop('survived', axis=1)
# y = titanic.frame['survived']
```

Use `ColumnTransformer` by selecting column by names

We will train our classifier with the following features:

Numeric Features:

- age: float;
- fare: float.

Categorical Features:

- embarked: categories encoded as strings {'C', 'S', 'Q'};
- sex: categories encoded as strings {'female', 'male'};
- pclass: ordinal integers {1, 2, 3}.

We create the preprocessing pipelines for both numeric and categorical data. Note that `pclass` could either be treated as a categorical or numeric feature.

```
numeric_features = ['age', 'fare']
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())])

categorical_features = ['embarked', 'sex', 'pclass']
categorical_transformer = OneHotEncoder(handle_unknown='ignore')

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)])

# Append classifier to preprocessing pipeline.
# Now we have a full prediction pipeline.
clf = Pipeline(steps=[('preprocessor', preprocessor),
                      ('classifier', LogisticRegression())])

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                    random_state=0)

clf.fit(X_train, y_train)
print("model score: %.3f" % clf.score(X_test, y_test))
```

```
Out: model score: 0.790
```

HOW TO USE DIFFERENT TRANSFORMERS FOR DIFFERENT COLUMNS

Defining dtypes for mixed tranformers automatically

Select columns in pd.DataFrame by their dtypes using skilearn selector

df.info()

Used to fiund out how the columns will be categorized with the selecto

make_column_selector()

Can select columns based on

(i) Pandas data frame dtypes

(ii) or the columns name with a regex.

CAUTION: When using multiple selection criteria, **all** criteria must match for a column to be selected.

add i. supported Pandas df dtypes:

- ***numeric:*** *use "numeric"* , np.number, 'number'
- **categorical:** use "category"
- **object:** use object - this included str, and all other columns encoded as object (may be of any dtype)
- **datetimes:** use np.datetime64, 'datetime' or 'datetime64'
- **timedeltas:** use np.timedelta64, 'timedelta' or 'timedelta64'
- **datetimetz**: use 'datetimetz', or 'datetime64[ns, tz]'

```
>>> from sklearn.compose
...     import make_column_selector
...     as selector
>>> preprocessor = ColumnTransformer([
....     (num, numeric_transformer,
....     Selector(dtype_include="Category"),
```

MAPRAMETERS:

- **Pattern**
- **dtype_include**
- **dtype_exclude**

Example with selecting categorical and numerical dtypes

Use `ColumnTransformer` by selecting column by data types

When dealing with a cleaned dataset, the preprocessing can be automatic by using the data types of the column to decide whether to treat a column as a numerical or categorical feature. `sklearn.compose.make_column_selector` gives this possibility. First, let's only select a subset of columns to simplify our example.

```
subset_feature = ['embarked', 'sex', 'pclass', 'age', 'fare']
X_train, X_test = X_train[subset_feature], X_test[subset_feature]
```

Then, we introspect the information regarding each column data type.

```
X_train.info()
```

```
Out: <class 'pandas.core.frame.DataFrame'>
     Int64Index: 1047 entries, 1118 to 684
     Data columns (total 5 columns):
      #   Column    Non-Null Count  Dtype
     ---  ------    --------------  -----
      0   embarked  1045 non-null   category
      1   sex       1047 non-null   category
      2   pclass    1047 non-null   float64
      3   age       841 non-null    float64
      4   fare      1046 non-null   float64
     dtypes: category(2), float64(3)
     memory usage: 35.0 KB
```

We can observe that the `embarked` and `sex` columns were tagged as `category` columns when loading the data with `fetch_openml`. Therefore, we can use this information to dispatch the categorical columns to the `categorical_transformer` and the remaining columns to the `numerical_transformer`.

Note: In practice, you will have to handle yourself the column data type. If you want some columns to be considered as `category`, you will have to convert them into categorical columns. If you are using pandas, you can refer to their documentation regarding Categorical data.

```
from sklearn.compose import make_column_selector as selector

preprocessor = ColumnTransformer(transformers=[
    ('num', numeric_transformer, selector(dtype_exclude="category")),
    ('cat', categorical_transformer, selector(dtype_include="category"))
])
clf = Pipeline(steps=[('preprocessor', preprocessor),
                      ('classifier', LogisticRegression())])

clf.fit(X_train, y_train)
print("model score: %.3f" % clf.score(X_test, y_test))
```

```
Out: model score: 0.794
```

The resulting score is not exactly the same as the one from the previous pipeline because the dtype-based selector treats the `pclass` column as a numeric feature instead of a categorical feature as previously:

```
selector(dtype_exclude="category")(X_train)
```

```
Out: ['pclass', 'age', 'fare']
```

```
selector(dtype_include="category")(X_train)
```

```
Out: ['embarked', 'sex']
```

sparse matrices vs numpy arrays

https://stackoverflow.com/questions/36969886/using-a-sparse-matrix-versus-numpy-array

HOW TO VISUALIZE THE PIPELINE

set_configs()

Use function that set global scikit-learn configuration And ask for displaying nice diagram, instead of text

CAUTION: it changes all global settings!

```
>>> from sklearn import set_config
>>> set_config(display="diagram")
    # "text" is an alternative, and
      normally used.
>>> pipeline
```

CLICK